Jay Alammar 杰伊·阿拉马尔 (/)

Visualizing machine learning one concept at a time.

一次可视化一个机器学习概念。

Read our book, Hands-On Large Language Models (https://www.LLM-book.com) and follow me on LinkedIn (https://www.linkedin.com/in/jalammar/), Bluesky (https://bsky.app/profile/jayalammar.bsky.social), Substack (https://newsletter.languagemodels.co/), X (https://x.com/JayAlammar),YouTube (https://www.youtube.com/channel/UCmOwsoHty5PrmE-3QhUBfPQ)

阅读我们的书《大型语言模型实践 (https://www.LLM-book.com)》，并在领英 (https://www.linkedin.com/in/jalammar/)、Bluesky (https://bsky.app/profile/jayalammar.bsky.social)、Substack (https://newsletter.languagemodels.co/)、X (https://x.com/JayAlammar)、YouTube (https://www.youtube.com/channel/UCmOwsoHty5PrmE-3QhUBfPQ)上关注我。

The Illustrated Transformer 图解Transformer

Discussions: Hacker News (65 points, 4 comments) (https://news.ycombinator.com/item?id=18351674), Reddit r/MachineLearning (29 points, 3 comments) (https://www.reddit.com/r/MachineLearning/comments/8uh2yz/p_the_illustrated_transformer_a_visual_look_at/)

讨论：黑客新闻（65分，4条评论）（https://news.ycombinator.com/item?id=18351674），红迪网机器学习板块（29分，3条评论）(https://www.reddit.com/r/MachineLearning/comments/8uh2yz/p_the_illustrated_transformer_a_visual_look_at/)

Translations: Arabic (https://www.mundhor.site/post/post14), Chinese (Simplified) 1 (https://blog.csdn.net/yujianmin1990/article/details/85221271), Chinese (Simplified) 2 (https://blog.csdn.net/qq_36667170/article/details/124359818), French 1 (https://a-coles.github.io/2020/11/15/transformer-illustre.html), French 2 (https://lbourdois.github.io/blog/nlp/Transformer/), Italian (https://medium.com/@val.mannucci/il-transformer-illustrato-it-37a78e3e2348), Japanese (https://tips-memo.com/translation-jayalmmar-transformer), Korean (https://nlpinkorean.github.io/illustrated-transformer/), Persian (http://dml.qom.ac.ir/2022/05/17/illustrated-transformer/), Russian (https://habr.com/ru/post/486358/), Spanish 1 (https://www.ibidemgroup.com/edu/transformer-ilustrado-jay-alammar/), Spanish 2 (https://hackernoon.com/el-transformador-ilustrado-una-traduccion-al-espanol-0y73wwp), Vietnamese (https://trituenhantao.io/tin-tuc/minh-hoa-transformer/)

翻译：阿拉伯语 (https://www.mundhor.site/post/post14)、中文（简体）1 (https://blog.csdn.net/yujianmin1990/article/details/85221271)、中文（简体）2 (https://blog.csdn.net/qq_36667170/article/details/124359818)、法语1 (https://a-coles.github.io/2020/11/15/transformer-illustre.html)、法语2 (https://lbourdois.github.io/blog/nlp/Transformer/)、意大利语 (https://medium.com/@val.mannucci/il-transformer-illustrato-it-37a78e3e2348)、日语 (https://tips-memo.com/translation-jayalmmar-transformer)、韩语 (https://nlpinkorean.github.io/illustrated-transformer/)、波斯语 (http://dml.qom.ac.ir/2022/05/17/illustrated-transformer/)、俄语 (https://habr.com/ru/post/486358/)、西班牙语1 (https://www.ibidemgroup.com/edu/transformer-ilustrado-jay-alammar/)、西班牙语2 (https://hackernoon.com/el-transformador-ilustrado-una-traduccion-al-espanol-0y73wwp)、越南语 (https://trituenhantao.io/tin-tuc/minh-hoa-transformer/)

Watch: MIT's Deep Learning State of the Art (https://youtu.be/53YvP6gdD7U?t=432) lecture referencing this post

观看：麻省理工学院的深度学习前沿技术 (https://youtu.be/53YvP6gdD7U?t=432)讲座引用了本文

Featured in courses at Stanford (https://web.stanford.edu/class/cs224n/), Harvard (https://scholar.harvard.edu/binxuw/classes/machine-learning-scratch/materials/transformers), MIT (https://ocw.mit.edu/courses/6-s897-machine-learning-for-healthcare-spring-2019/d39a6eed387ee90b1f72a01949c35c7b_MIT6_S897S19_lec8.pdf), Princeton (https://www.cs.princeton.edu/courses/archive/fall22/cos597G/), CMU (https://www.cs.cmu.edu/~leili/course/mldl22w/14-Transformer.pdf) and others

在斯坦福大学、哈佛大学、麻省理工学院、普林斯顿大学、卡内基梅隆大学等院校的课程中被重点介绍

(https://www.llm-book.com)

Update: This post has now become a book! Check out LLM-book.com (https://llm-book.com) which contains (Chapter 3) an updated and expanded version of this post speaking about the latest Transformer models and how they've evolved in the seven years since the original Transformer (like Multi-Query Attention and RoPE Positional embeddings).

更新：这篇文章现在已经成书了！请查看LLM-book.com (https://llm-book.com)，其中（第3章）包含了这篇文章的更新和扩展版本，讲述了最新的Transformer模型以及自原始Transformer问世七年来它们的发展（比如多查询注意力机制和RoPE位置嵌入）。

In the previous post, we looked at Attention (https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/) – a ubiquitous method in modern deep learning models. Attention is a concept that helped improve the performance of neural machine translation applications. In this post, we will look at **The Transformer** – a model that uses attention to boost the speed with which these models can be trained. The Transformer outperforms the Google Neural Machine Translation model in specific tasks. The biggest benefit, however, comes from how The Transformer lends itself to parallelization. It is in fact Google Cloud's recommendation to use The Transformer as a reference model to use their Cloud TPU (https://cloud.google.com/tpu/) offering. So let's try to break the model apart and look at how it functions.

在上一篇文章中，我们探讨了注意力机制——这是现代深度学习模型中一种普遍使用的方法。注意力机制这一概念有助于提升神经机器翻译应用的性能。在本文中，我们将介绍Transformer——一种利用注意力机制来提高模型训练速度的模型。在特定任务中，Transformer的表现优于谷歌神经机器翻译模型。然而，其最大的优势在于Transformer易于并行化。事实上，谷歌云建议将Transformer作为参考模型，以使用他们的Cloud TPU产品。那么，让我们试着剖析这个模型，看看它是如何运作的。

The Transformer was proposed in the paper Attention is All You Need (https://arxiv.org/abs/1706.03762). A TensorFlow implementation of it is available as a part of the Tensor2Tensor (https://github.com/tensorflow/tensor2tensor) package. Harvard's NLP group created a guide annotating the paper with PyTorch implementation (http://nlp.seas.harvard.edu/2018/04/03/attention.html). In this post, we will attempt to oversimplify things a bit and introduce the concepts one by one to hopefully make it easier to understand to people without in-depth knowledge of the subject matter.

Transformer 是在论文《Attention is All You Need (https://arxiv.org/abs/1706.03762)》中提出的。其 TensorFlow 实现可作为 Tensor2Tensor (https://github.com/tensorflow/tensor2tensor) 包的一部分获取。哈佛大学的自然语言处理小组创建了一份 指南，其中用 PyTorch 实现对该论文进行了注释 (http://nlp.seas.harvard.edu/2018/04/03/attention.html)。在本文中，我们将尝试对内容进行一些简化，并逐一介绍相关概念，希望能让没有该领域深入知识的人更容易理解。

2025 Update: We've built a free short course (https://bit.ly/4aRnn7Z) that brings the contents of this post up-to-date with animations:

2025年更新：我们制作了一门免费短期课程 (https://bit.ly/4aRnn7Z)，通过动画形式更新了本文的内容：

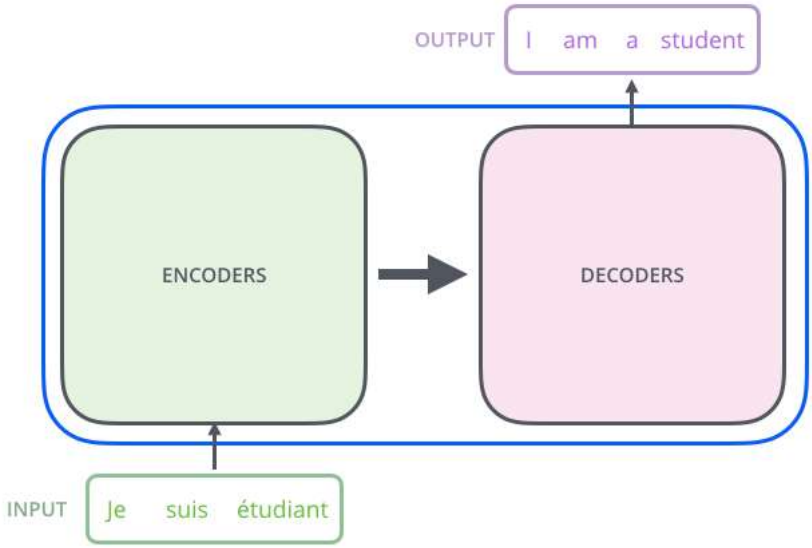
A High-Level Look 概览

Let's begin by looking at the model as a single black box. In a machine translation application, it would take a sentence in one language, and output its translation in another.

让我们先将该模型视为一个单一的黑箱。在机器翻译应用中，它会接收一种语言的句子，并输出其另一种语言的翻译。

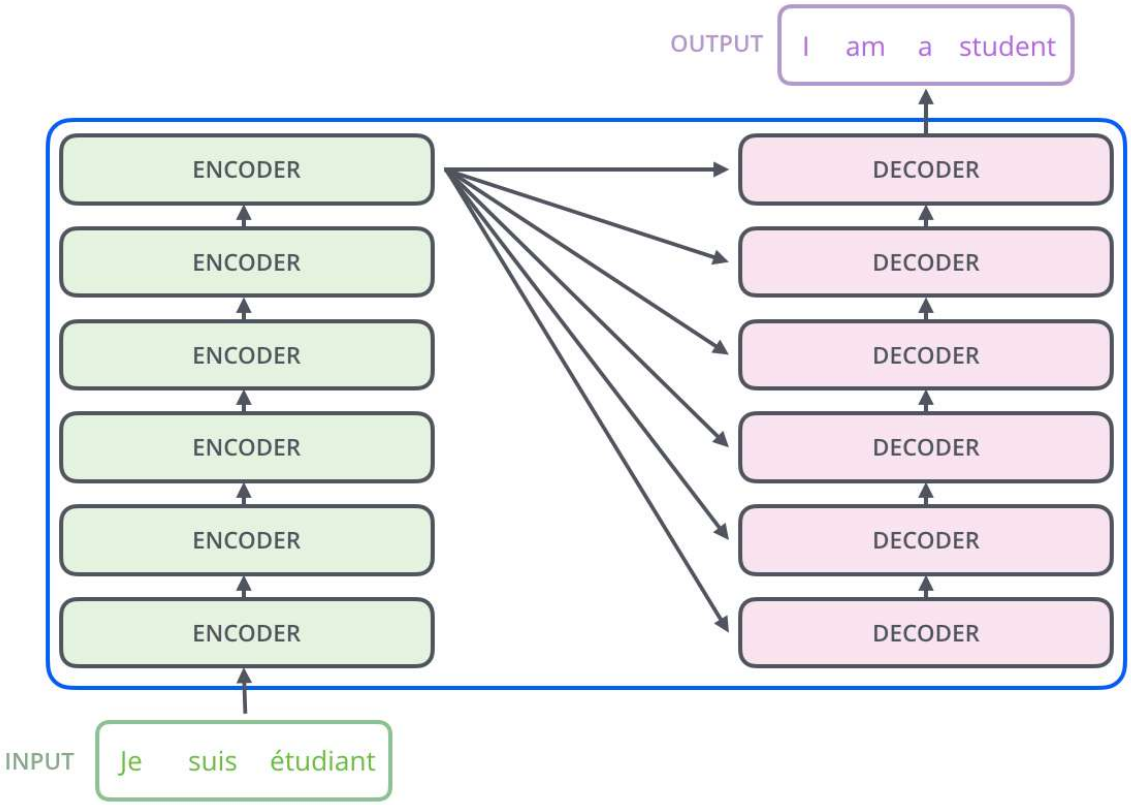
Popping open that Optimus Prime goodness, we see an encoding component, a decoding component, and connections between them.

打开擎天柱的核心部分，我们可以看到一个编码组件、一个解码组件以及它们之间的连接。



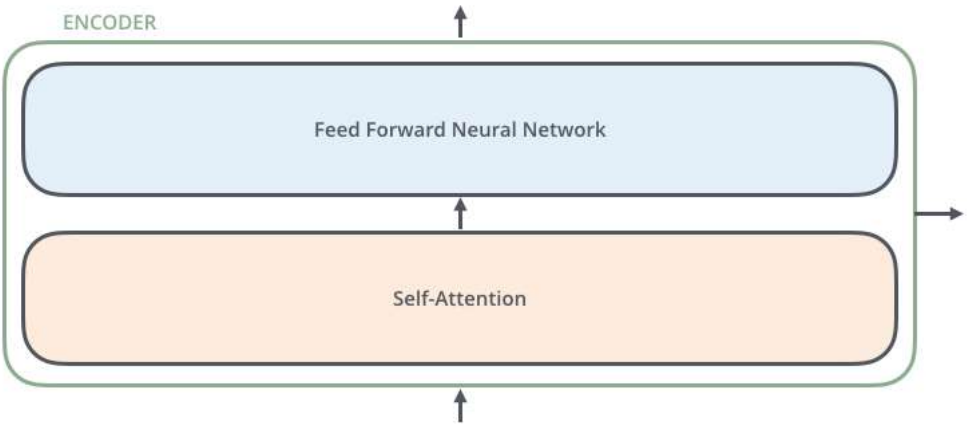
The encoding component is a stack of encoders (the paper stacks six of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements). The decoding component is a stack of decoders of the same number.

编码部分是一堆编码器的叠加（论文中叠加了六个——数字六并没有什么神奇之处，人们完全可以尝试其他组合）。解码部分则是数量相同的一堆解码器的叠加。



The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:

编码器在结构上完全相同（但它们不共享权重）。每个编码器都分为两个子层：



The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. We'll look closer at self-attention later in the post.

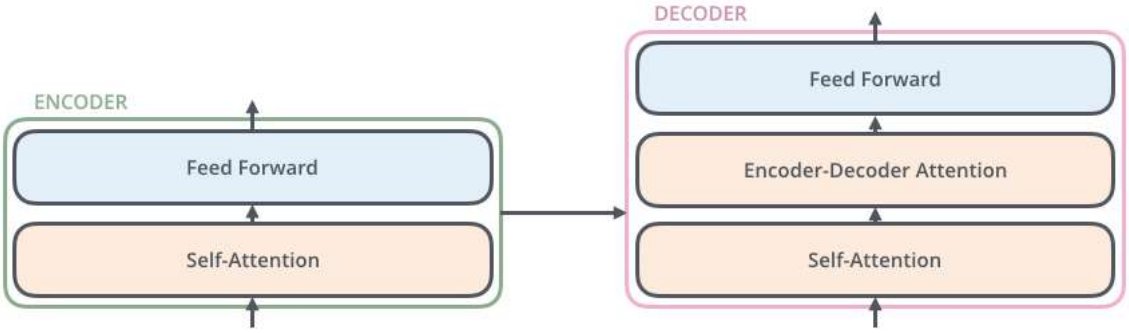
编码器的输入首先流经一个自注意力层——该层能帮助编码器在对特定单词进行编码时查看输入句子中的其他单词。我们将在本文后面更深入地探讨自注意力。

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

自注意力层的输出被输入到前馈神经网络中。完全相同的前馈网络被独立地应用于每个位置。

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in seq2seq models (https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/)).

解码器同时具备这两种层，但在它们之间有一个注意力层，该层帮助解码器专注于输入句子的相关部分（类似于序列到序列模型 (https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/)中注意力的作用）。



Bringing The Tensors Into The Picture

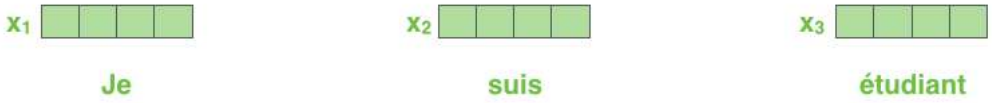
将张量引入视野

Now that we've seen the major components of the model, let's start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output.

既然我们已经了解了该模型的主要组成部分，接下来就让我们看看各种向量/张量，以及它们如何在这些组件之间流动，从而将训练好的模型的输入转化为输出。

As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm (https://medium.com/deeper-learning/glossary-of-deep-learning-word-embedding-f90c3cec34ca).

与一般的自然语言处理应用程序一样，我们首先使用一种嵌入算法 (https://medium.com/deeper-learning/glossary-of-deep-learning-word-embedding-f90c3cec34ca)将每个输入词转换为向量。



Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

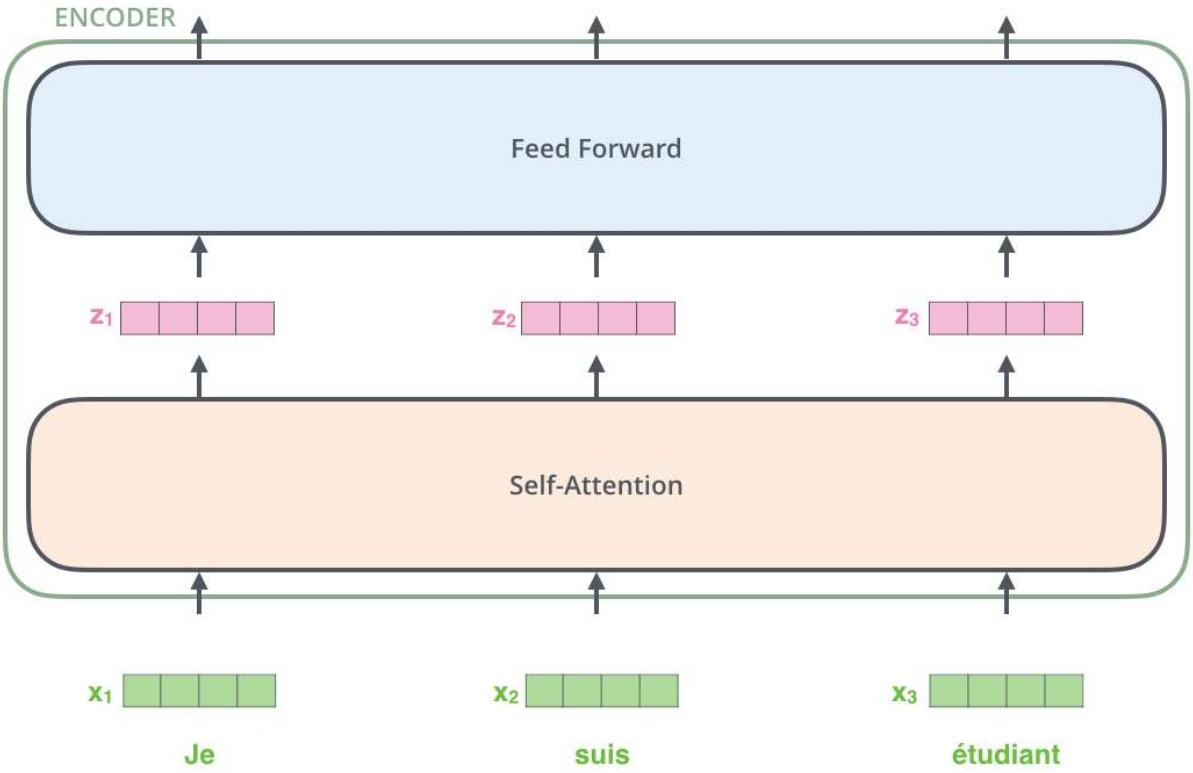
每个单词都被嵌入到一个512维的向量中。我们将用这些简单的方框来表示这些向量。

The embedding only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512 – In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. The size of this list is hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.

嵌入过程只发生在最底层的编码器中。所有编码器共有的抽象特征是，它们都会接收一个向量列表，每个向量的大小为512——在最底层的编码器中，这些向量是词嵌入；而在其他编码器中，这些向量则是其正下方编码器的输出。这个列表的大小是我们可以设置的超参数——基本上，它会是我们训练数据集中最长句子的长度。

After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.

在对输入序列中的词语进行嵌入处理后，每个词语都会流经编码器的两层网络。



Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer. The feed-forward layer does not have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.

在这里，我们开始看到Transformer的一个关键特性，即每个位置的词都在编码器中沿着自己的路径流动。在自注意力层中，这些路径之间存在依赖关系。然而，前馈层没有这些依赖关系，因此在流经前馈层时，不同的路径可以并行执行。

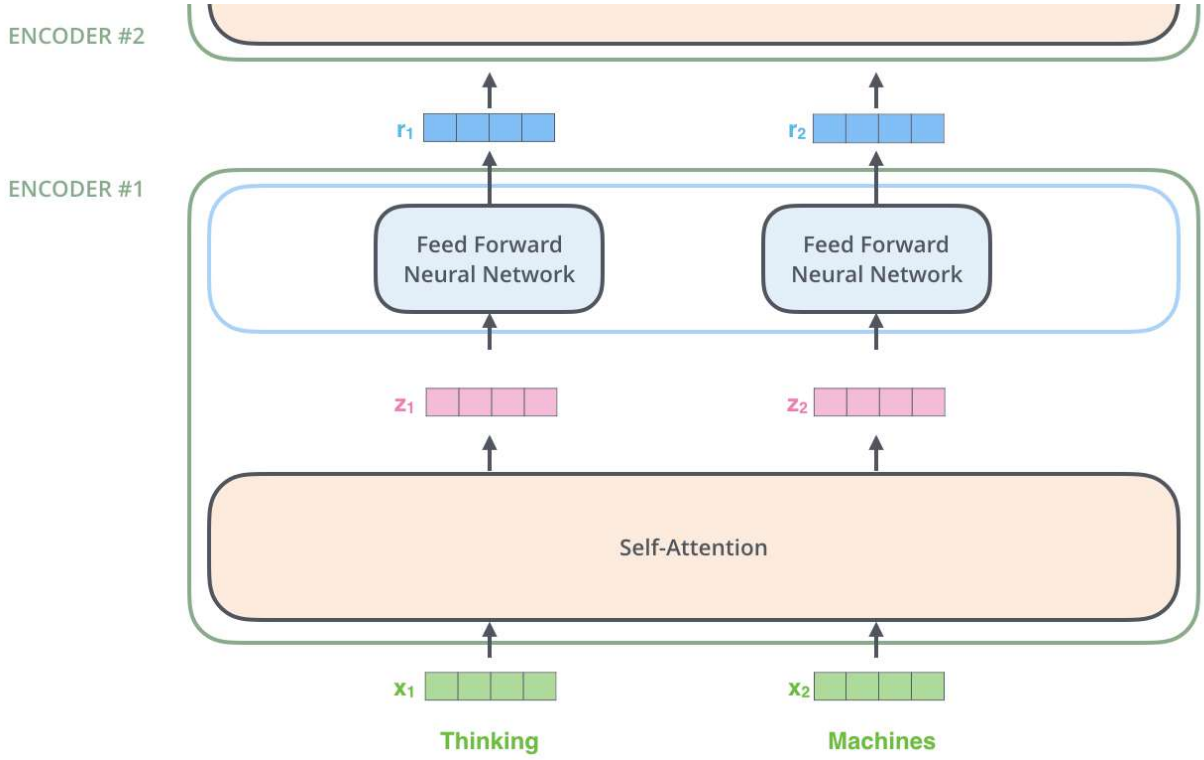
Next, we'll switch up the example to a shorter sentence and we'll look at what happens in each sub-layer of the encoder.

接下来，我们将换一个更短的句子作为例子，并观察编码器的每个子层中会发生什么。

Now We're Encoding! 现在我们开始编码！

As we've mentioned already, an encoder receives a list of vectors as input. It processes this list by passing these vectors into a 'self-attention' layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.

正如我们已经提到的，编码器接收一个向量列表作为输入。它通过将这些向量传入“自注意力”层、然后传入前馈神经网络来处理这个列表，之后将输出向上发送到下一个编码器。



The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.

每个位置上的词都会经过一个自注意力过程。然后，它们各自通过一个前馈神经网络——每个向量分别流经的是完全相同的网络。

Self-Attention at a High Level 高层级的自注意力机制

Don’t be fooled by me throwing around the word “self-attention” like it’s a concept everyone should be familiar with. I had personally never came across the concept until reading the Attention is All You Need paper. Let us distill how it works.

别被我随口说出“自注意力”这个词给骗了，好像这是个每个人都应该熟悉的概念似的。我自己也是在读《Attention is All You Need》这篇论文时才第一次接触到这个概念。让我们来好好理解一下它的工作原理。

Say the following sentence is an input sentence we want to translate:

假设下面这句话是我们想要翻译的输入句：

```
”The animal didn't cross the street because it was too tired”
```

```
“The animal didn't cross the street because it was too tired”
```

What does “it” in this sentence refer to? Is it referring to the street or to the animal? It’s a simple question to a human, but not as simple to an algorithm.

这句话中的“it”指的是什么？是指街道还是动物？这对人类来说是个简单的问题，但对算法来说就没那么简单了。

When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.

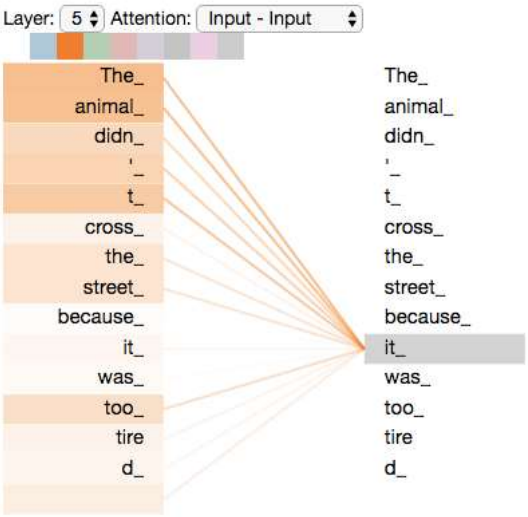
当模型处理“it”这个词时，自注意力机制使其能够将“it”与“animal”联系起来。

As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

当模型处理每个词（输入序列中的每个位置）时，自注意力机制使其能够查看输入序列中的其他位置，以寻找有助于更好地编码该词的线索。

If you’re familiar with RNNs, think of how maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it’s processing. Self-attention is the method the Transformer uses to bake the “understanding” of other relevant words into the one we’re currently processing.

如果你熟悉循环神经网络（RNN），不妨想想维持隐藏状态是如何让RNN将其对已处理过的先前词语/向量的表示与当前正在处理的内容相结合的。自注意力机制是Transformer所采用的一种方法，它能将对其他相关词语的“理解”融入到我们当前正在处理的词语中。



As we are encoding the word "it" in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of "it".

当我们在第5个编码器（栈顶的编码器）中对单词“it”进行编码时，注意力机制的一部分会聚焦于“The Animal”，并将其部分表示融入到“it”的编码中。

Be sure to check out the Tensor2Tensor notebook (https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb) where you can load a Transformer model, and examine it using this interactive visualization.

一定要查看Tensor2Tensor笔记本(https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb)，在那里你可以加载Transformer模型，并使用这种交互式可视化进行检查。

Self-Attention in Detail 详细的自注意力机制

Let’s first look at how to calculate self-attention using vectors, then proceed to look at how it’s actually implemented – using matrices.

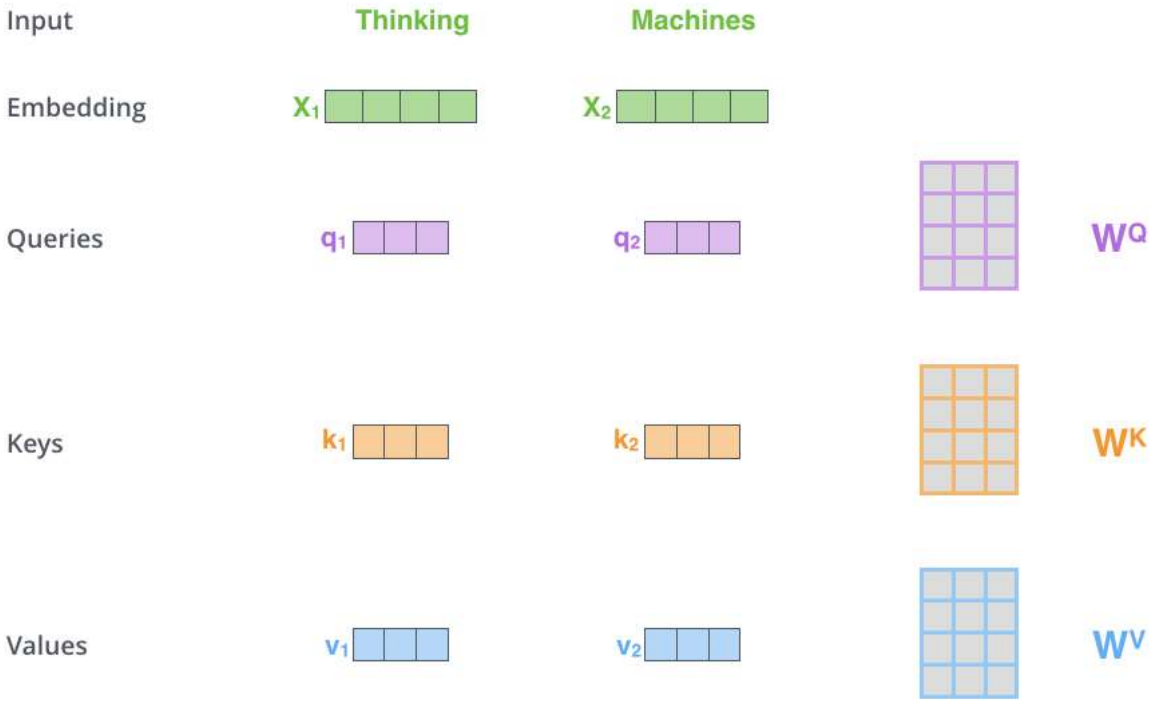
让我们首先看看如何使用向量计算自注意力，然后再看看它实际上是如何实现的——使用矩阵。

The **first step** in calculating self-attention is to create three vectors from each of the encoder’s input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

计算自注意力的第一步是从编码器的每个输入向量（在这种情况下，是每个单词的嵌入向量）中创建三个向量。因此，对于每个单词，我们会创建一个查询向量、一个键向量和一个值向量。这些向量是通过将嵌入向量与我们在训练过程中训练的三个矩阵相乘而得到的。

Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512. They don’t HAVE to be smaller, this is an architecture choice to make the computation of multiheaded attention (mostly) constant.

请注意，这些新向量的维度小于嵌入向量。它们的维度是64，而嵌入向量和编码器的输入/输出向量的维度是512。它们并非必须更小，这是一种架构选择，目的是使多头注意力的计算（在很大程度上）保持恒定。



Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

将x1与WQ权重矩阵相乘会得到q1，即与该词相关联的“查询”向量。最终，我们为输入句子中的每个词创建了“查询”“键”和“值”投影。

What are the “query”, “key”, and “value” vectors?

“查询”“键”和“值”向量是什么？

They’re abstractions that are useful for calculating and thinking about attention. Once you proceed with reading how attention is calculated below, you’ll know pretty much all you need to know about the role each of these vectors plays.

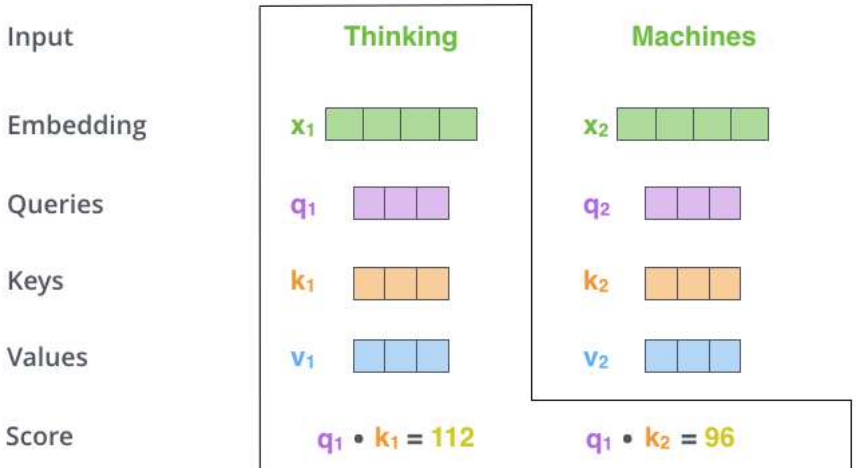
它们是有助于计算和理解注意力的抽象概念。一旦你继续阅读下面关于注意力如何计算的内容，你就会大致了解这些向量各自所扮演的角色。

The **second step** in calculating self-attention is to calculate a score. Say we’re calculating the self-attention for the first word in this example, “Thinking”. We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

计算自注意力的**第二步**是计算分数。假设我们要计算本例中第一个词“Thinking”的自注意力。我们需要将输入句子中的每个词与该词进行评分。这个分数决定了我们在对特定位置的词进行编码时，要将多少注意力放在输入句子的其他部分。

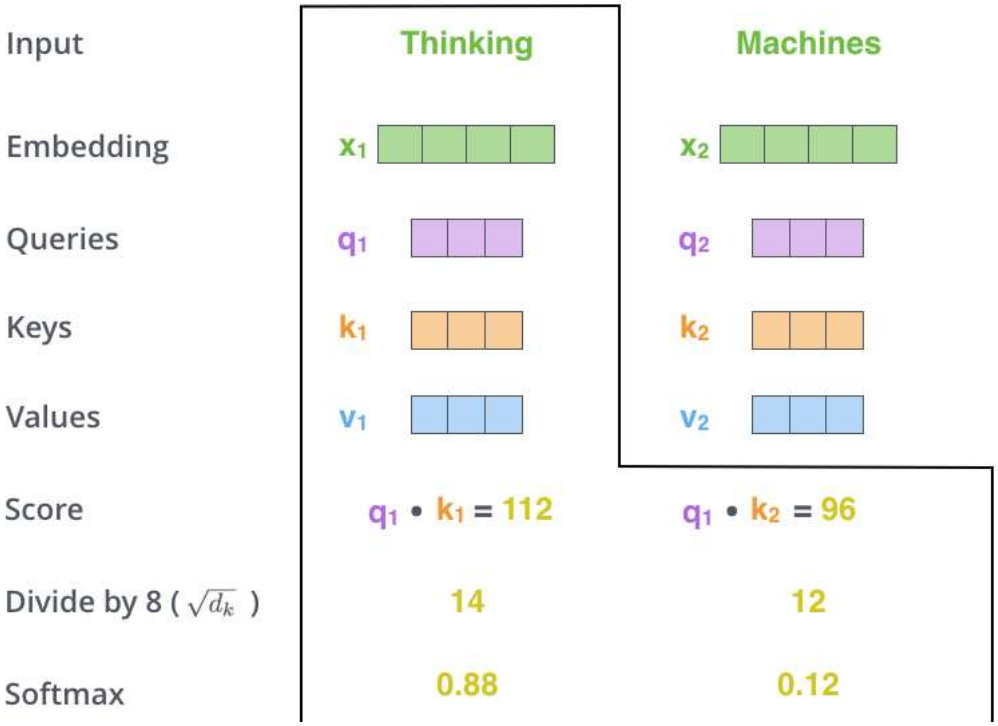
The score is calculated by taking the dot product of the query vector with the key vector of the respective word we’re scoring. So if we’re processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.

分数的计算方式是取查询向量与我们要评分的相应单词的键向量的点积。因此，如果我们正在处理位置#1处单词的自注意力，第一个分数就是q1和k1的点积，第二个分数则是q1和k2的点积。



The **third and fourth steps** are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.

第三步和第四步是将分数除以8（这是论文中使用的关键向量维度的平方根——64。这样做会使梯度更稳定。这里也可以使用其他可能的值，但这是默认值），然后将结果通过softmax运算。Softmax会对分数进行归一化，使其均为正值且总和为1。



This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

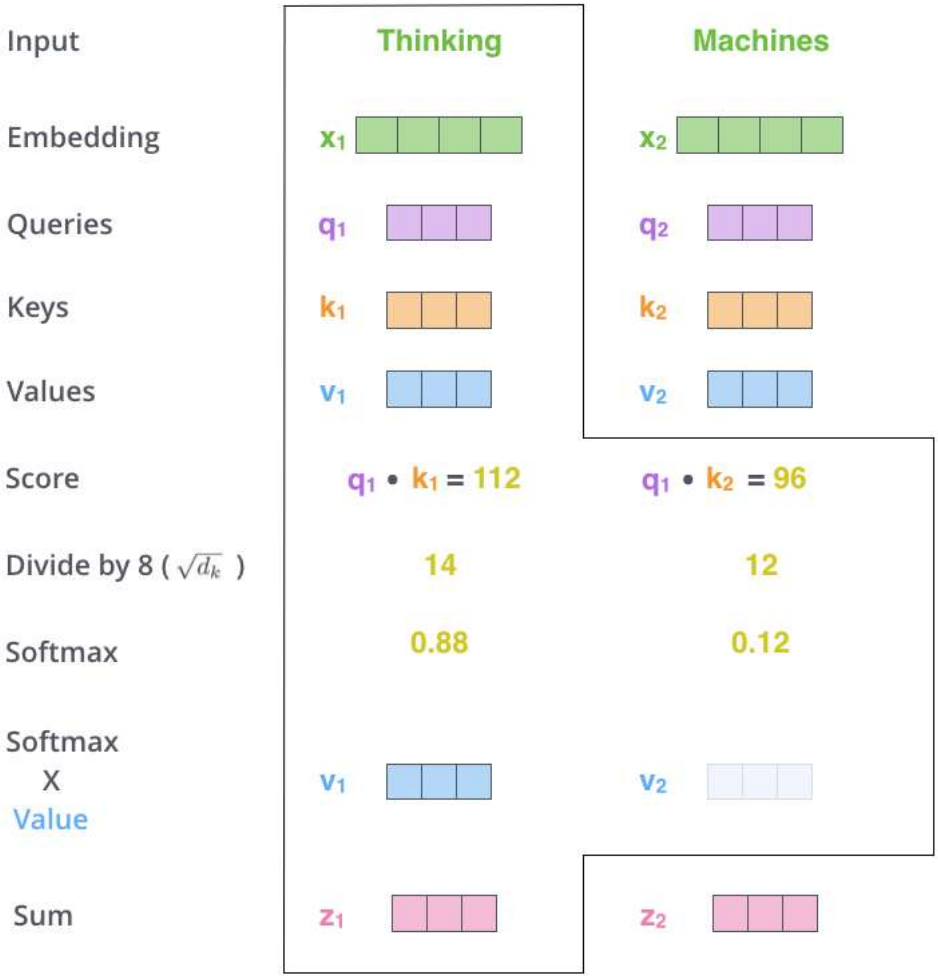
这个softmax分数决定了每个词在该位置的表达程度。显然，该位置的词会有最高的softmax分数，但有时关注与当前词相关的另一个词会很有用。

The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

第五步是将每个值向量乘以softmax分数（为求和做准备）。这里的思路是保持我们想要关注的词的值不变，而淹没不相关的词（例如，通过将它们乘以像0.001这样的小数）。

The **sixth step** is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

第六步是对加权值向量求和。这就产生了自注意力层在该位置（针对第一个词）的输出。



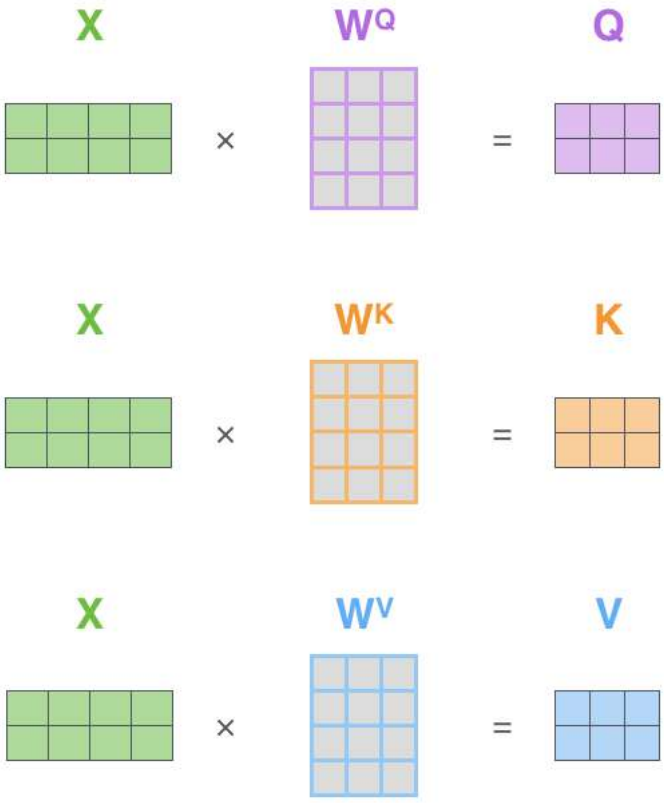
That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is done in matrix form for faster processing. So let's look at that now that we've seen the intuition of the calculation on the word level.

以上就是自注意力计算的全部内容。得到的向量可以传递给前馈神经网络。不过，在实际实现中，为了加快处理速度，这种计算是以矩阵形式进行的。既然我们已经了解了在单词层面上计算的原理，现在就让我们来看看矩阵形式的计算吧。

Matrix Calculation of Self-Attention 自注意力的矩阵计算

The first step is to calculate the Query, Key, and Value matrices. We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (WQ, WK, WV).

第一步是计算查询（Query）、键（Key）和值（Value）矩阵。我们通过将嵌入向量打包成矩阵X，并将其与我们训练好的权重矩阵（WQ、WK、WV）相乘来实现这一点。

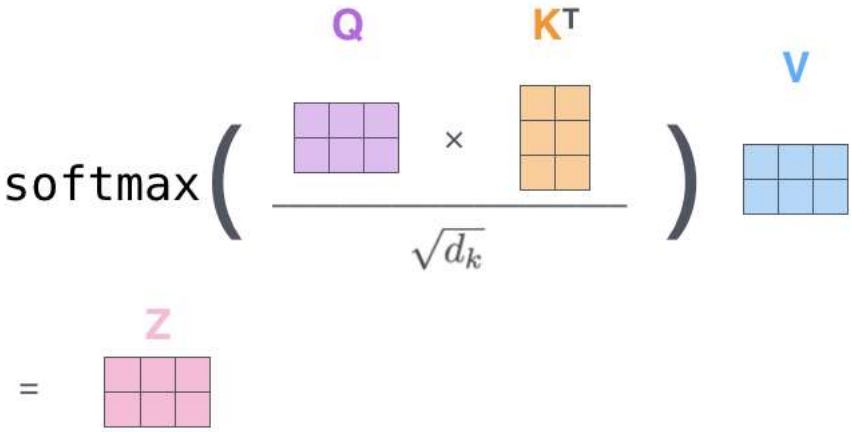


Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

X矩阵中的每一行对应输入句子中的一个单词。我们再次看到嵌入向量（512维，或图中的4个方框）和q/k/v向量（64维，或图中的3个方框）在大小上的差异。

Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

最后，由于我们处理的是矩阵，我们可以将第二步到第六步浓缩成一个公式，以计算自注意力层的输出。



The self-attention calculation in matrix form

矩阵形式的自注意力计算

The Beast With Many Heads 多头巨兽

The paper further refined the self-attention layer by adding a mechanism called "multi-headed" attention. This improves the performance of the attention layer in two ways:

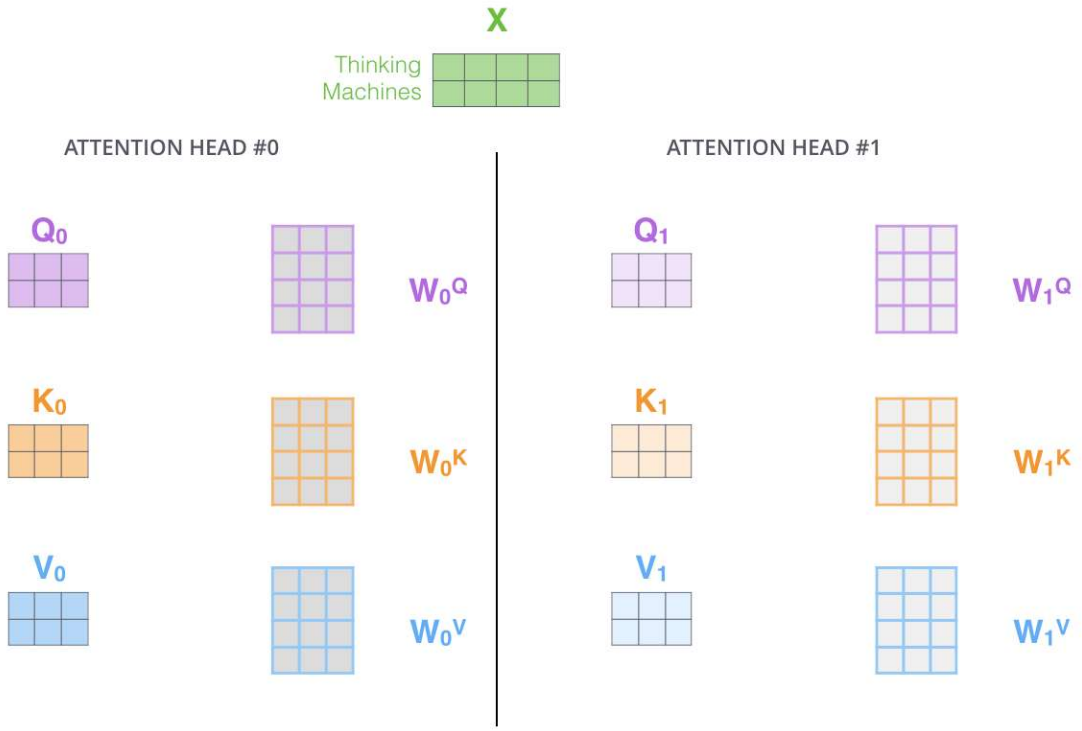
该论文通过添加一种名为"多头"注意力的机制，进一步优化了自注意力层。这从两个方面提升了注意力层的性能：

1. It expands the model's ability to focus on different positions. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the actual word itself. If we're translating a sentence like "The animal didn't cross the street because it was too tired", it would be useful to know which word "it" refers to.

   它增强了模型关注不同位置的能力。是的，在上面的例子中，z1包含了其他所有编码的一小部分，但它可能主要由实际的单词本身决定。如果我们要翻译像"那只动物没有过马路，因为它太累了"这样的句子，知道"它"指的是哪个词会很有用。

2. It gives the attention layer multiple "representation subspaces". As we'll see next, with multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.

   它为注意力层提供了多个"表示子空间"。接下来我们会看到，在多头注意力机制中，我们拥有的不仅仅是一组，而是多组查询/键/值权重矩阵（Transformer使用八个注意力头，因此每个编码器/解码器最终会有八组）。这些矩阵组中的每一组都是随机初始化的。然后，在训练之后，每一组都会被用来将输入嵌入（或来自较低层编码器/解码器的向量）投影到不同的表示子空间中。

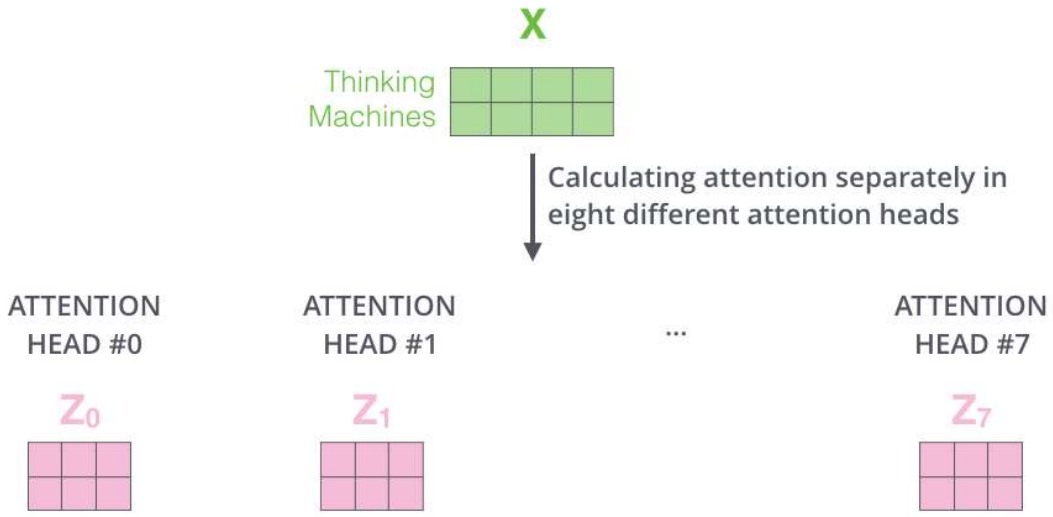


With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

通过多头注意力机制，我们为每个头保留独立的Q/K/V权重矩阵，从而得到不同的Q/K/V矩阵。和之前一样，我们将X与WQ/WK/WV矩阵相乘，以生成Q/K/V矩阵。

If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices

如果我们按照上面概述的方法进行相同的自注意力计算，只是使用不同的权重矩阵进行八次不同的计算，最终会得到八个不同的Z矩阵。

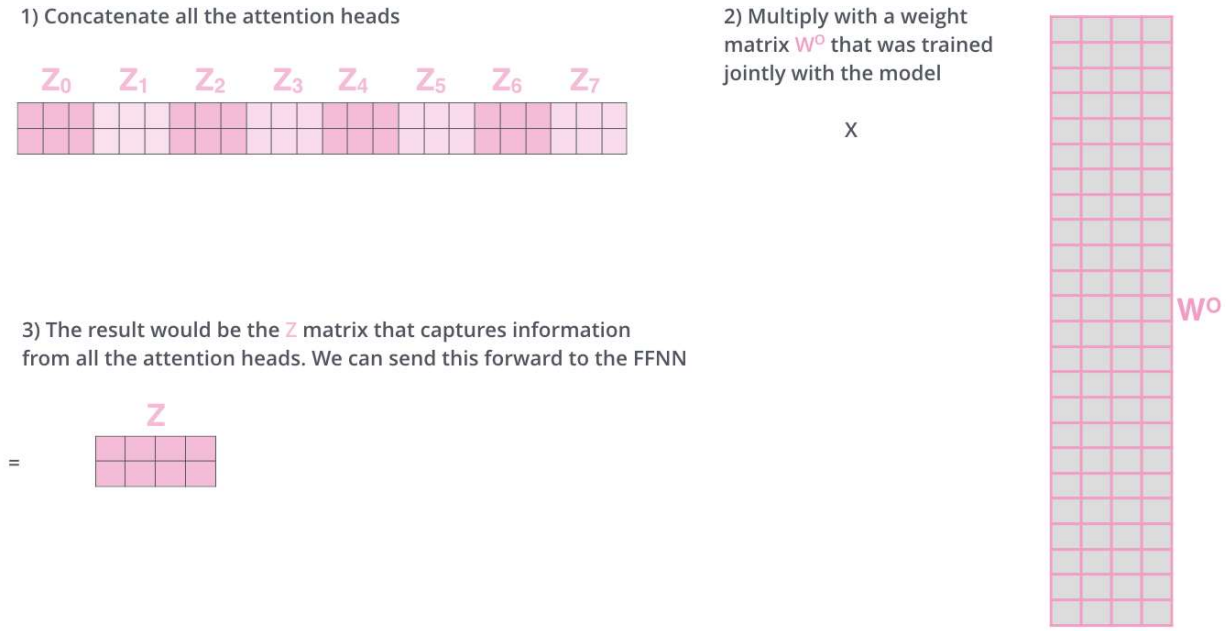


This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.

这给我们带来了一点挑战。前馈层并不期望得到八个矩阵——它期望的是一个单一矩阵（每个词对应一个向量）。因此，我们需要一种方法将这八个矩阵压缩成一个单一矩阵。

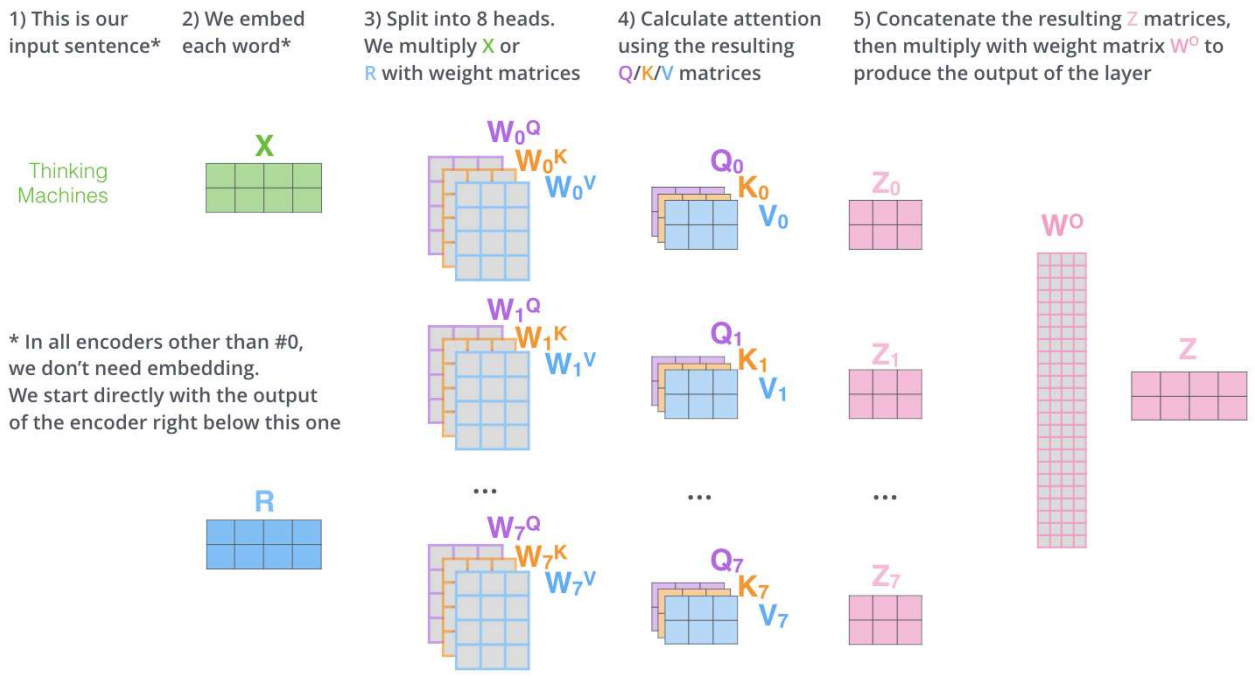
How do we do that? We concat the matrices then multiply them by an additional weights matrix WO.

我们是如何做到的呢？我们将这些矩阵连接起来，然后将它们与一个额外的权重矩阵WO相乘。



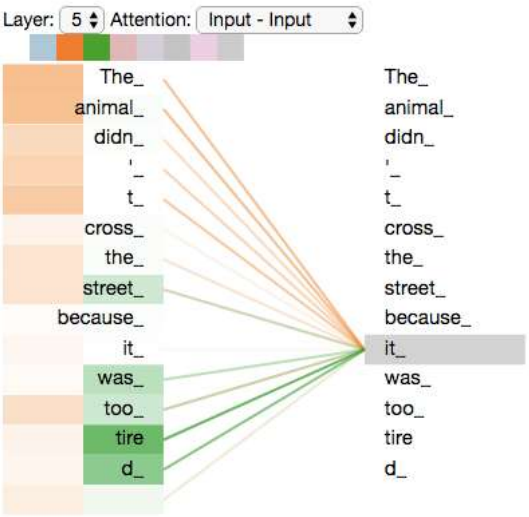
That's pretty much all there is to multi-headed self-attention. It's quite a handful of matrices, I realize. Let me try to put them all in one visual so we can look at them in one place

这差不多就是多头自注意力机制的全部内容了。我知道这里面涉及到不少矩阵。我来试着把它们都整合到一个可视化图表中，这样我们就能在一个地方看到所有矩阵了。



Now that we have touched upon attention heads, let's revisit our example from before to see where the different attention heads are focusing as we encode the word "it" in our example sentence:

既然我们已经谈及注意力头，那就让我们回顾一下之前的例子，看看在对例句中的"it"进行编码时，不同的注意力头都在关注哪些内容：

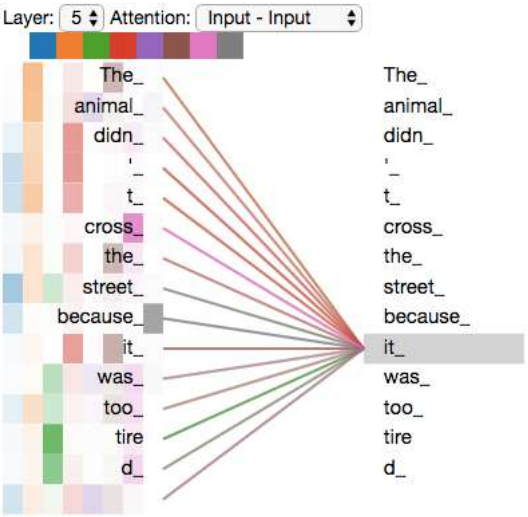


As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

当我们对"it"这个词进行编码时，一个注意力头主要关注"the animal"，而另一个注意力头则关注"tired"——从某种意义上说，模型对"it"这个词的表征融入了"animal"和"tired"两者的部分表征。

If we add all the attention heads to the picture, however, things can be harder to interpret:

然而，如果我们把所有注意力头都纳入考量，情况就会更难解释：



Representing The Order of The Sequence Using Positional Encoding

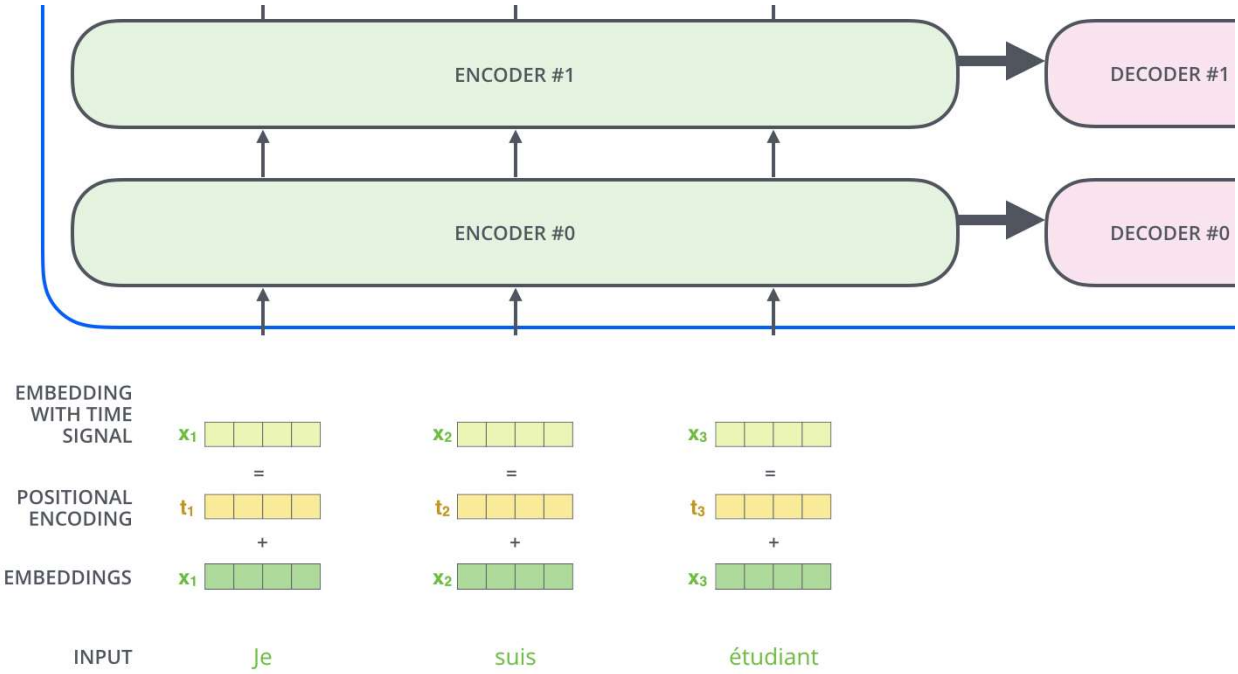
使用位置编码表示序列的顺序

One thing that's missing from the model as we have described it so far is a way to account for the order of the words in the input sequence.

到目前为止，我们所描述的模型中缺少的一点是，它没有办法考虑输入序列中词语的顺序。

To address this, the transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence. The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.

为了解决这个问题，Transformer会向每个输入嵌入添加一个向量。这些向量遵循模型学习到的特定模式，这有助于模型确定每个词的位置，或者序列中不同词之间的距离。这里的思路是，将这些值添加到嵌入中后，在将嵌入向量投影到Q/K/V向量时以及在点积注意力过程中，能使嵌入向量之间产生有意义的距离。

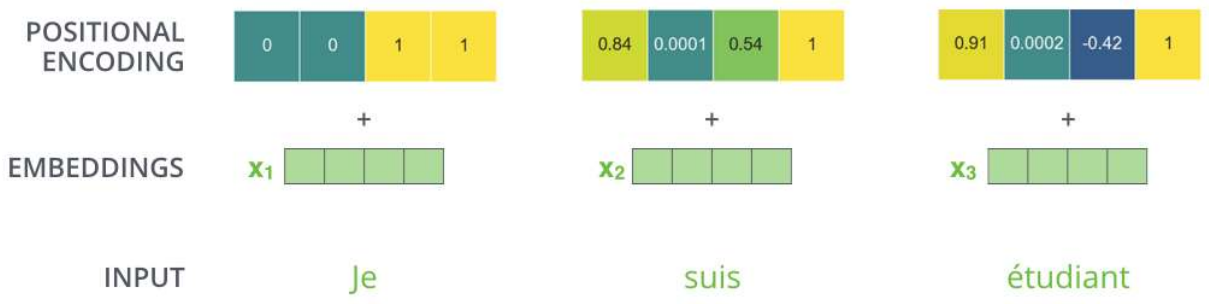


To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.

为了让模型理解词语的顺序，我们添加了位置编码向量——其值遵循特定的模式。

If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like this:

如果我们假设嵌入的维度为4，那么实际的位置编码会是这样的：



A real example of positional encoding with a toy embedding size of 4

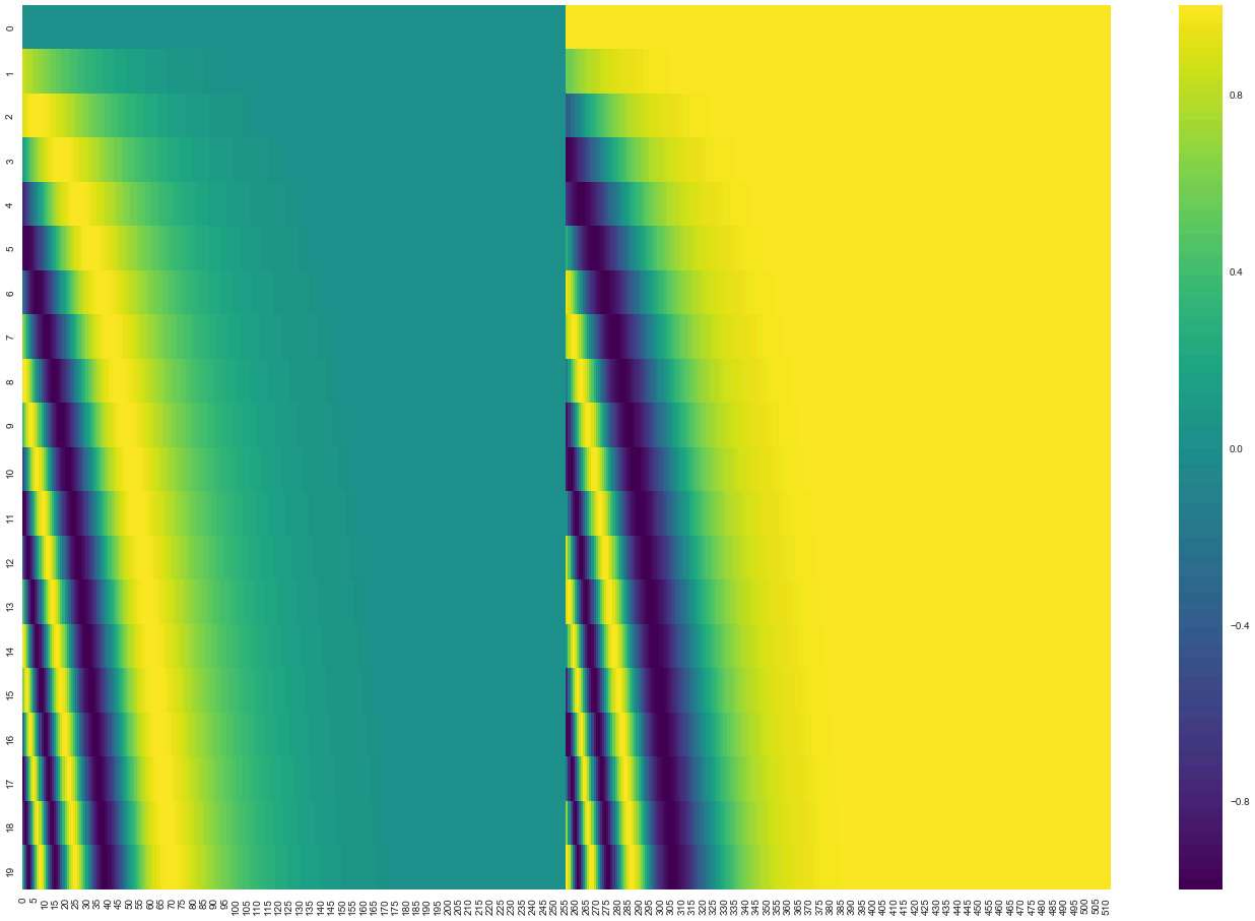
嵌入维度为4的位置编码实例

What might this pattern look like?

这种模式可能是什么样子的?

In the following figure, each row corresponds to a positional encoding of a vector. So the first row would be the vector we'd add to the embedding of the first word in an input sequence. Each row contains 512 values – each with a value between 1 and -1. We've color-coded them so the pattern is visible.

在下图中，每一行对应一个向量的位置编码。因此，第一行就是我们要添加到输入序列中第一个单词嵌入的向量。每行包含512个值，每个值都在1到-1之间。我们对这些值进行了颜色编码，以便清晰地看到其模式。



A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in half down the center. That's because the values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). They're then concatenated to form each of the positional encoding vectors.

一个包含20个单词（行）、嵌入维度为512（列）的位置编码实例。你可以看到它在中间位置被分成了两半。这是因为左半部分的值由一个函数（使用正弦函数）生成，而右半部分则由另一个函数（使用余弦函数）生成。然后它们被拼接起来，形成每个位置编码向量。

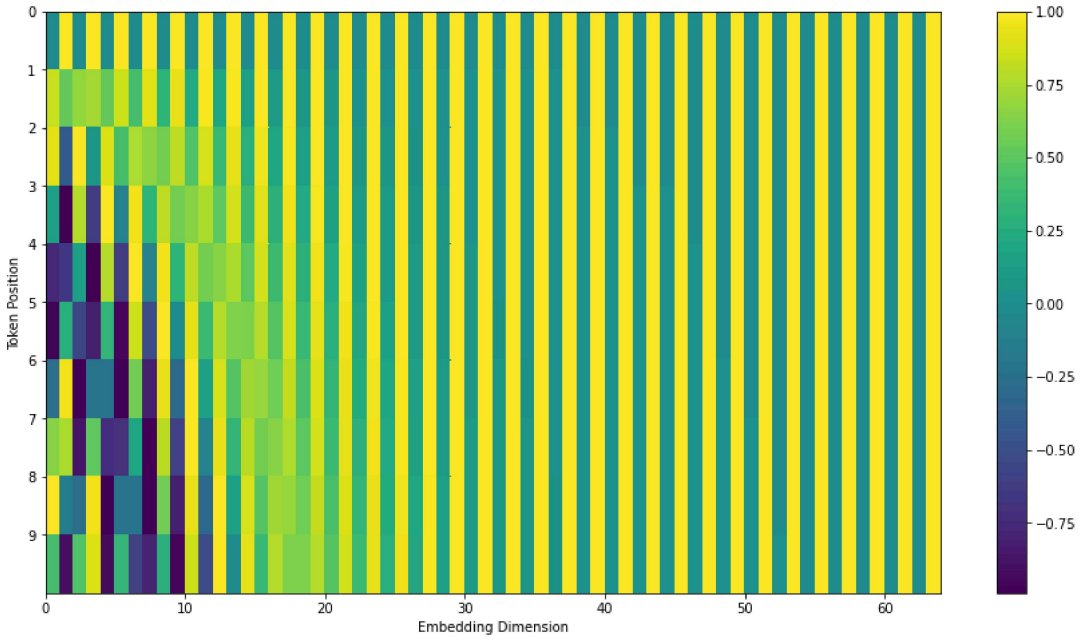
The formula for positional encoding is described in the paper (section 3.5). You can see the code for generating positional encodings in `get_timing_signal_1d()` (https://github.com/tensorflow/tensor2tensor/blob/23bd23b9830059fbc349381b70d9429b5c40a139/tensor2tensor/layers/comm

This is not the only possible method for positional encoding. It, however, gives the advantage of being able to scale to unseen lengths of sequences (e.g. if our trained model is asked to translate a sentence longer than any of those in our training set).

位置编码的公式在论文（3.5节）中有描述。你可以在`get_timing_signal_1d()` (https://github.com/tensorflow/tensor2tensor/blob/23bd23b9830059fbc349381b70d9429b5c40a139/tensor2tensor/layers/comm 中看到生成位置编码的代码。这并非唯一可行的位置编码方法。不过，它的优势在于能够扩展到未见过的序列长度（例如，如果我们训练好的模型被要求翻译一个比训练集中任何句子都长的句子）。

July 2020 Update: The positional encoding shown above is from the Tensor2Tensor implementation of the Transformer. The method shown in the paper is slightly different in that it doesn't directly concatenate, but interweaves the two signals. The following figure shows what that looks like. Here's the code to generate it (https://github.com/jalammar/jalammar.github.io/blob/master/notebookes/transformer/transformer_positional_encoding_graph.ip

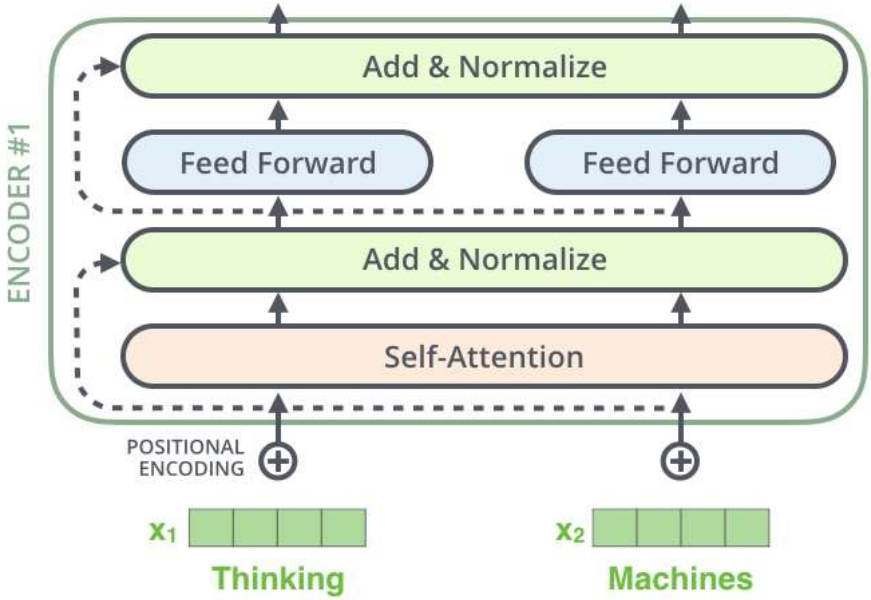
2020年7月更新： 上面显示的位置编码来自Transformer的Tensor2Tensor实现。论文中展示的方法略有不同，它不直接进行拼接，而是将两个信号交织在一起。下图展示了其具体样子。以下是生成它的代码 (https://github.com/jalammar/jalammar.github.io/blob/master/notebookes/transformer/transformer_positional_encoding_graph.ip



The Residuals 残差

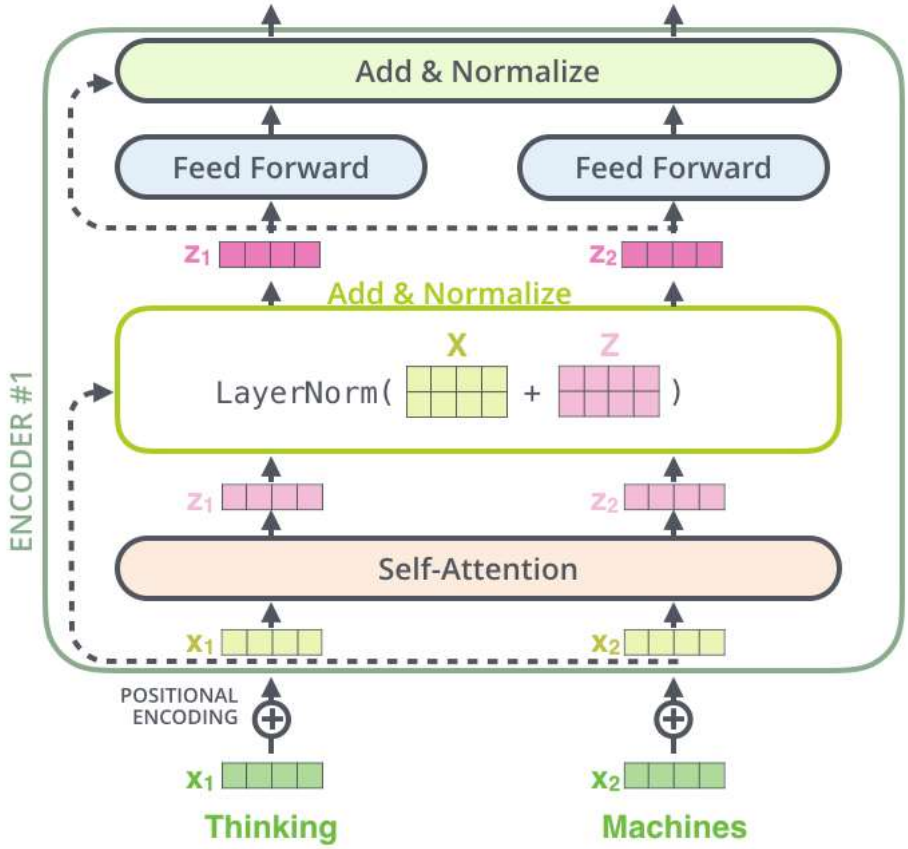
One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization (https://arxiv.org/abs/1607.06450) step.

在继续之前，我们需要提及编码器架构中的一个细节，即每个编码器中的每个子层（自注意力、前馈神经网络）周围都有一个残差连接，并且其后会跟着一个层归一化 (https://arxiv.org/abs/1607.06450)步骤。



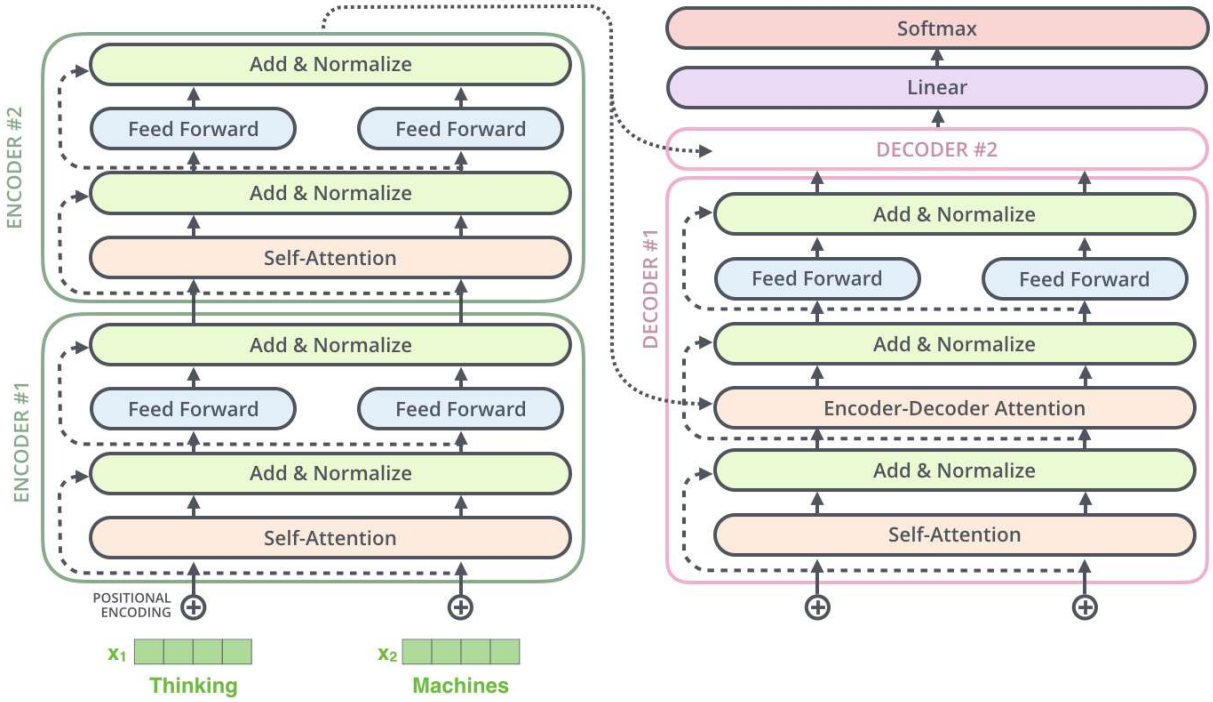
If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:

如果我们要将与自注意力相关的向量和层归一化操作可视化，它会是这样的：



This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:

这一点也适用于解码器的子层。如果我们想象一个由2个堆叠的编码器和解码器组成的Transformer，它大概会是这个样子：



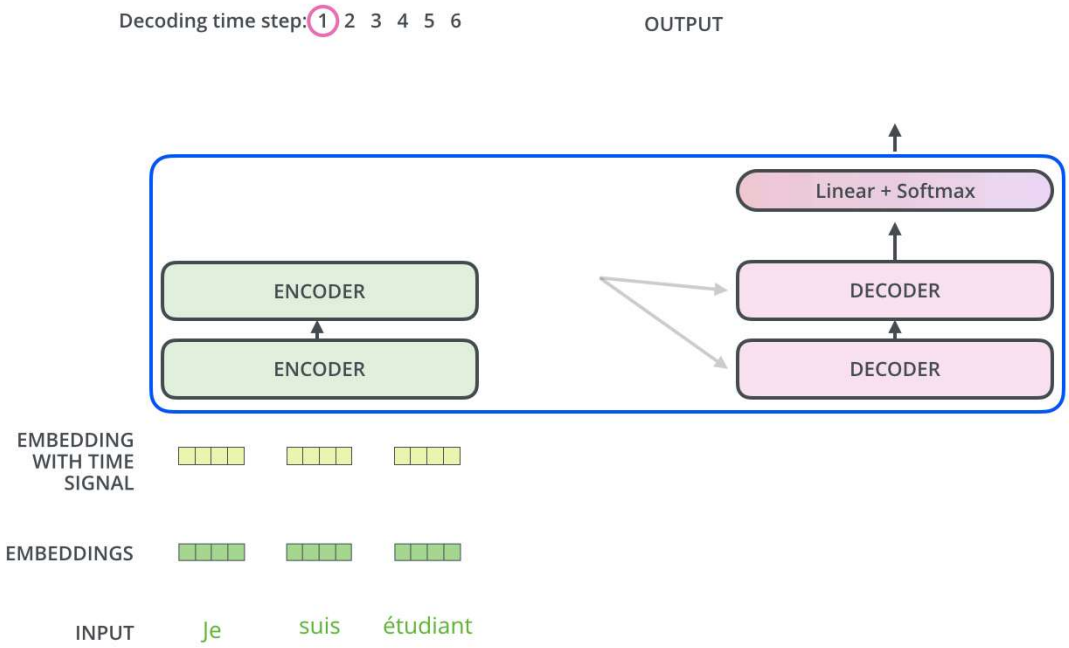
The Decoder Side 解码器端

Now that we've covered most of the concepts on the encoder side, we basically know how the components of decoders work as well. But let's take a look at how they work together.

既然我们已经涵盖了编码器方面的大部分概念，那么我们基本上也了解了解码器各组件的工作原理。不过，让我们来看看它们是如何协同工作的。

The encoder start by processing the input sequence. The output of the top encoder is then transformed into a set of attention vectors K and V. These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence:

编码器首先对输入序列进行处理。顶部编码器的输出随后被转换为一组注意力向量K和V。这些向量将被每个解码器在其"编码器-解码器注意力"层中使用，该层帮助解码器将注意力集中在输入序列的适当位置：

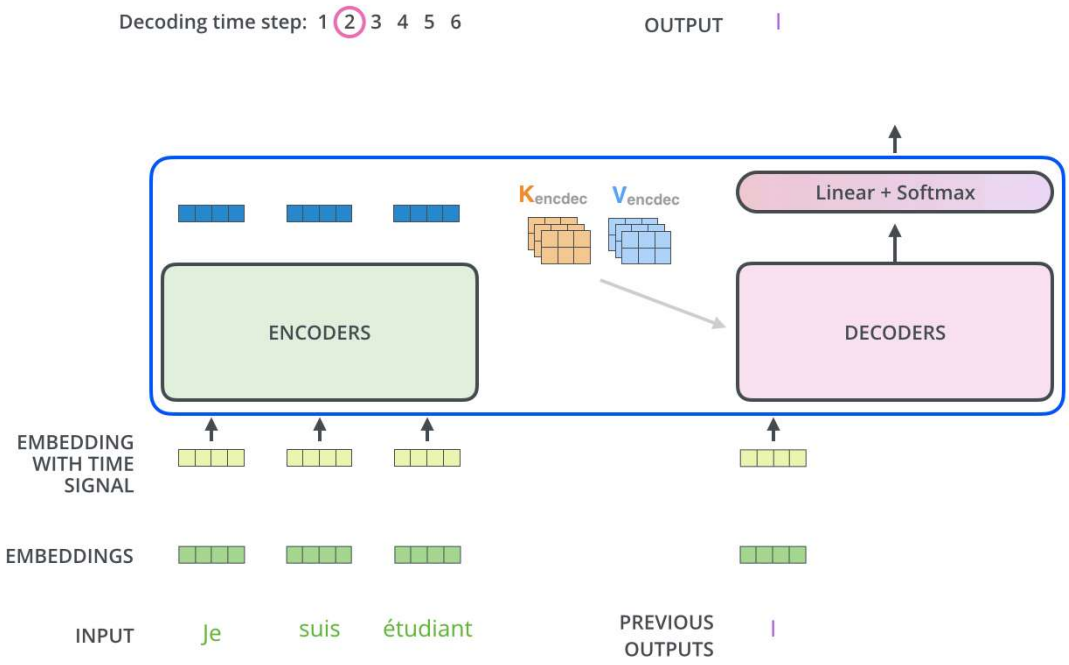


After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

完成编码阶段后，我们开始解码阶段。解码阶段的每一步都会从输出序列（在本例中为英语翻译句子）中输出一个元素。

The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.

以下步骤会重复该过程，直到遇到一个特殊符号，该符号表明Transformer解码器已完成其输出。每一步的输出会作为输入馈送到下一时间步的底层解码器，解码器会像编码器那样向上传递其解码结果。并且，与处理编码器输入的方式类似，我们会对这些解码器输入进行嵌入处理并添加位置编码，以指示每个单词的位置。



The self attention layers in the decoder operate in a slightly different way than the one in the encoder:

解码器中的自注意力层与编码器中的自注意力层运作方式略有不同：

In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to `-inf`) before the softmax step in the self-attention calculation.

在解码器中，自注意力层只允许关注输出序列中较早的位置。这是通过在自注意力计算的softmax步骤之前对未来位置进行掩码（将它们设置为`-inf`）来实现的。

The “Encoder-Decoder Attention” layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

“编码器-解码器注意力”层的工作方式与多头自注意力类似，不同之处在于它从其下方的层创建查询矩阵，并从编码器堆栈的输出中获取键矩阵和值矩阵。

The Final Linear and Softmax Layer 最后的线性层和Softmax层

The decoder stack outputs a vector of floats. How do we turn that into a word? That’s the job of the final Linear layer which is followed by a Softmax Layer.

解码器堆栈输出一个浮点数向量。我们如何将其转换为一个单词呢？这就是最后一个线性层的工作，该线性层之后会接一个 Softmax 层。

The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

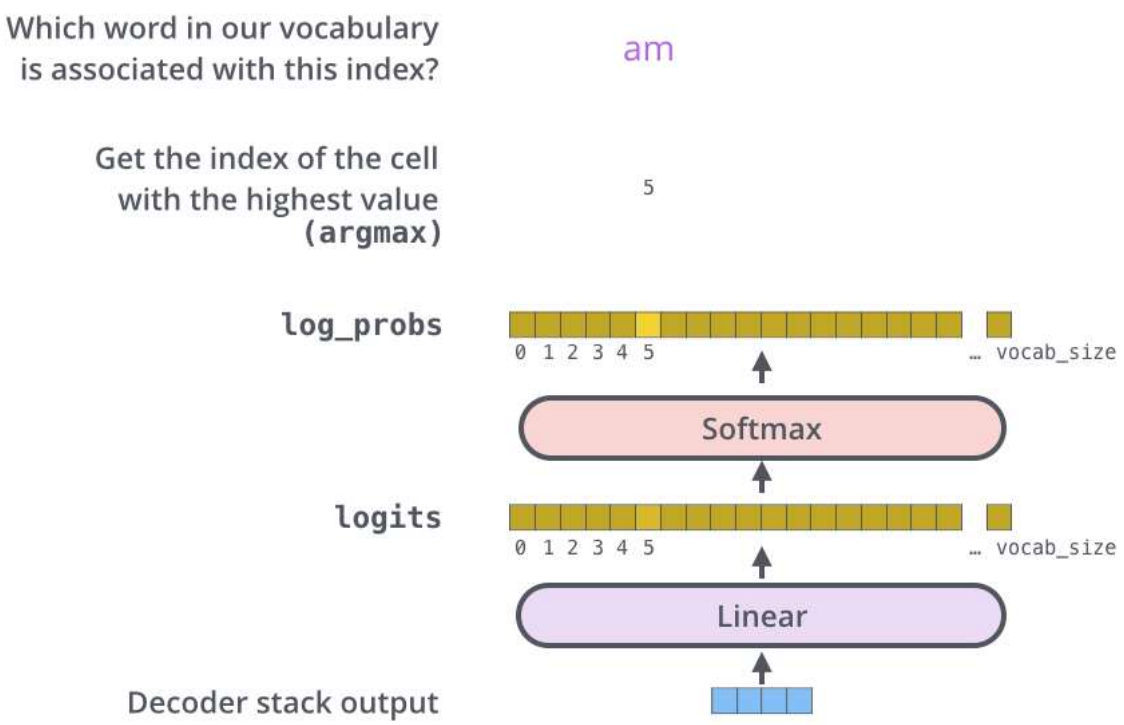
线性层是一个简单的全连接神经网络，它将解码器堆叠产生的向量投射到一个更大更大的向量中，这个向量被称为logits向量。

Let’s assume that our model knows 10,000 unique English words (our model’s “output vocabulary”) that it’s learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.

假设我们的模型认识从训练数据集中学到的10,000个独特的英语单词（即模型的“输出词汇表”）。这会使logits向量的宽度为10,000个单元——每个单元对应一个独特单词的得分。这就是我们对模型输出以及后续线性层的解读方式。

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

然后，softmax层将这些分数转换为概率（所有概率均为正数，且总和为1.0）。选择概率最高的单元格，与之关联的单词便作为该时间步的输出。



This figure starts from the bottom with the vector produced as the output of the decoder stack. It is then turned into an output word.

此图从底部开始，底部是解码器堆栈输出的向量。然后，该向量被转换为输出词。

Recap Of Training 训练回顾

Now that we've covered the entire forward-pass process through a trained Transformer, it would be useful to glance at the intuition of training the model.

既然我们已经介绍了经过训练的Transformer的整个前向传播过程，那么了解一下训练该模型的思路会很有帮助。

During training, an untrained model would go through the exact same forward pass. But since we are training it on a labeled training dataset, we can compare its output with the actual correct output.

在训练过程中，一个未经训练的模型会经历完全相同的前向传播。但由于我们是在带标签的训练数据集上对其进行训练，因此可以将其输出与实际的正确输出进行比较。

To visualize this, let's assume our output vocabulary only contains six words("a", "am", "i", "thanks", "student", and "<eos>" (short for 'end of sentence')).

为了直观理解这一点，我们假设输出词汇表只包含六个单词（"a"、"am"、"i"、"thanks"、"student"和"<eos>"（"句子结束"的缩写））。

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

The output vocabulary of our model is created in the preprocessing phase before we even begin training.

我们模型的输出词汇表是在预处理阶段创建的，甚至在我们开始训练之前就已经完成了。

Once we define our output vocabulary, we can use a vector of the same width to indicate each word in our vocabulary. This also known as one-hot encoding. So for example, we can indicate the word “am” using the following vector:

一旦我们定义了输出词汇表，就可以使用一个相同宽度的向量来表示词汇表中的每个词。这也被称为独热编码。例如，我们可以用以下向量来表示“am”这个词：

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word “am”

0.0	1.0	0.0	0.0	0.0	0.0

Example: one-hot encoding of our output vocabulary

示例：我们输出词汇表的独热编码

Following this recap, let’s discuss the model’s loss function – the metric we are optimizing during the training phase to lead up to a trained and hopefully amazingly accurate model.

在回顾之后，让我们来讨论模型的损失函数——这是我们在训练阶段进行优化的指标，目的是得到一个经过训练且希望能极其准确的模型。

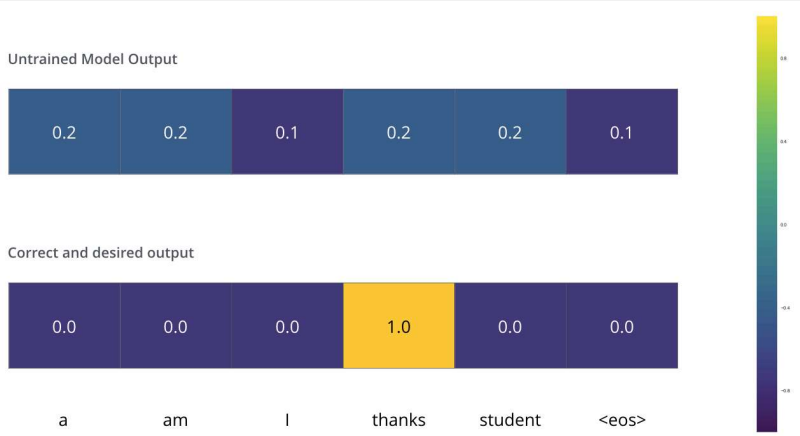
The Loss Function 损失函数

Say we are training our model. Say it’s our first step in the training phase, and we’re training it on a simple example – translating “merci” into “thanks”.

假设我们正在训练我们的模型。假设这是训练阶段的第一步，我们用一个简单的例子来训练它——将“merci”翻译成“谢谢”。

What this means, is that we want the output to be a probability distribution indicating the word “thanks”. But since this model is not yet trained, that’s unlikely to happen just yet.

这意味着，我们希望输出是一个表明“谢谢”这个词的概率分布。但由于该模型尚未经过训练，目前还不太可能实现这一点。



Since the model's parameters (weights) are all initialized randomly, the (untrained) model produces a probability distribution with arbitrary values for each cell/word. We can compare it with the actual output, then tweak all the model's weights using backpropagation to make the output closer to the desired output.

由于模型的参数（权重）都是随机初始化的，（未经训练的）模型会为每个单元/单词生成具有任意值的概率分布。我们可以将其与实际输出进行比较，然后使用反向传播调整模型的所有权重，使输出更接近期望输出。

How do you compare two probability distributions? We simply subtract one from the other. For more details, look at cross-entropy (https://colah.github.io/posts/2015-09-Visual-Information/) and Kullback–Leibler divergence (https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained).

如何比较两种概率分布？我们只需用一个减去另一个。更多细节，请查看交叉熵 (https://colah.github.io/posts/2015-09-Visual-Information/)和KL散度 (https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained)。

But note that this is an oversimplified example. More realistically, we'll use a sentence longer than one word. For example – input: "je suis étudiant" and expected output: "i am a student". What this really means, is that we want our model to successively output probability distributions where:

但请注意，这是一个过于简化的例子。更实际的情况是，我们会使用一个长于一个单词的句子。例如——输入："je suis étudiant"，预期输出："i am a student"。这真正意味着，我们希望我们的模型连续输出概率分布，其中：

- Each probability distribution is represented by a vector of width vocab_size (6 in our toy example, but more realistically a number like 30,000 or 50,000)

  每个概率分布都由一个宽度为词汇表大小的向量表示（在我们的示例中为6，但更实际的数字可能是30,000或50,000）

- The first probability distribution has the highest probability at the cell associated with the word "i"

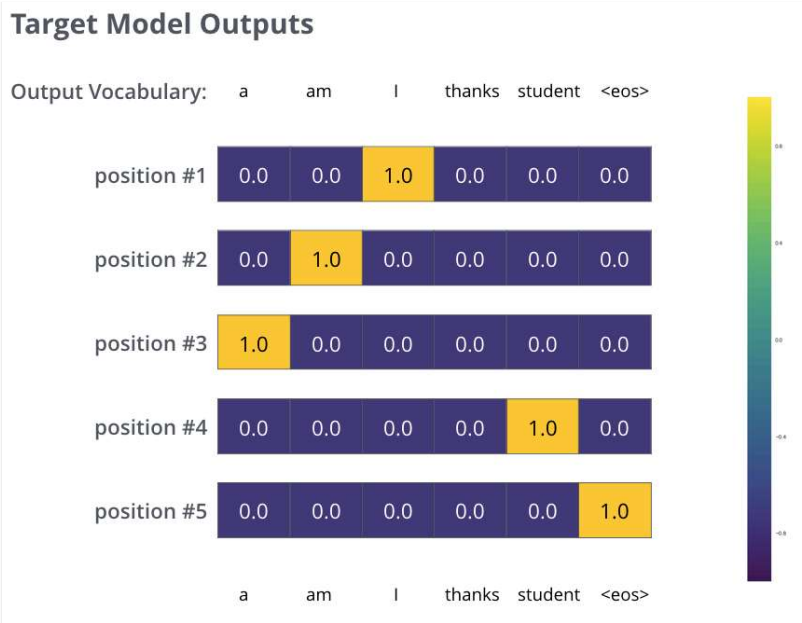
  第一个概率分布在与单词"i"相关的单元格处具有最高概率。

- The second probability distribution has the highest probability at the cell associated with the word "am"

  第二个概率分布在与单词"am"相关的单元格处具有最高概率。

- And so on, until the fifth output distribution indicates '`<end of sentence>`' symbol, which also has a cell associated with it from the 10,000 element vocabulary.

  依此类推，直到第五个输出分布显示"`<end of sentence>`"符号，该符号在包含10,000个元素的词汇表中也有一个与之关联的单元。

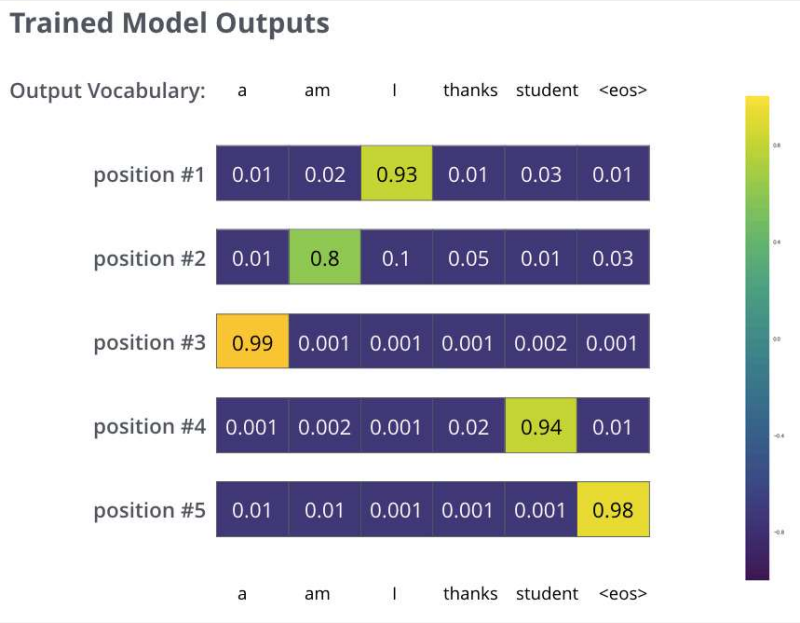


The targeted probability distributions we'll train our model against in the training example for one sample sentence.

在一个样本句子的训练示例中，我们将针对其训练模型的目标概率分布。

After training the model for enough time on a large enough dataset, we would hope the produced probability distributions would look like this:

在足够大的数据集上对模型进行足够长时间的训练后，我们希望生成的概率分布会是这样的：



Hopefully upon training, the model would output the right translation we expect. Of course it's no real indication if this phrase was part of the training dataset (see: cross validation (https://www.youtube.com/watch?v=TIgfjmp-4BA)). Notice that every position gets a little bit of probability even if it's unlikely to be the output of that time step -- that's a very useful property of softmax which helps the training process.

希望经过训练后，模型能输出我们期望的正确翻译。当然，如果这个短语是训练数据集的一部分，那就不能作为真正的参考了（参见：交叉验证 (https://www.youtube.com/watch?v=TIgfjmp-4BA)）。请注意，即使某个位置不太可能是该时间步的输出，它也会获得一点概率——这是softmax的一个非常有用的特性，有助于训练过程。

Now, because the model produces the outputs one at a time, we can assume that the model is selecting the word with the highest probability from that probability distribution and throwing away the rest. That's one way to do it (called greedy decoding). Another way to do it would be to hold on to, say, the top two words (say, 'I' and 'a' for example), then in the next step, run the model twice: once assuming the first output position was the word 'I', and another time assuming the first output position was the word 'a', and whichever version produced less error considering both positions #1 and #2 is kept. We repeat this for positions #2 and #3…etc. This method is called "beam search", where in our example, beam_size was two (meaning that at all times, two partial hypotheses (unfinished translations) are kept in memory), and top_beams is also two (meaning we'll return two translations). These are both hyperparameters that you can experiment with.

现在，由于模型一次生成一个输出，我们可以假设模型会从该概率分布中选择概率最高的词，而舍弃其余的词。这是一种实现方式（称为贪心解码）。另一种方式是保留，比如说，概率最高的两个词（例如"我"和"一个"），然后在下一步中，运行模型两次：一次假设第一个输出位置是词"我"，另一次假设第一个输出位置是词"一个"，然后保留综合考虑第一个和第二个位置时误差较小的那个版本。我们对第二个和第三个位置……以此类推，重复这个过程。这种方法被称为"波束搜索"，在我们的例子中，波束大小为2（意味着始终在内存中保留两个部分假设（未完成的翻译）），而顶级波束也为2（意味着我们将返回两个翻译结果）。这两个都是你可以进行试验的超参数。

Go Forth And Transform 勇往直前，实现变革

I hope you've found this a useful place to start to break the ice with the major concepts of the Transformer. If you want to go deeper, I'd suggest these next steps:

希望你已经发现这是一个了解Transformer主要概念的好起点。如果你想深入学习，我建议你采取以下步骤：

- Read the Attention Is All You Need (https://arxiv.org/abs/1706.03762) paper, the Transformer blog post (Transformer: A Novel Neural Network Architecture for Language Understanding (https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html)), and the Tensor2Tensor announcement (https://ai.googleblog.com/2017/06/accelerating-deep-learning-research.html).

  阅读《注意力就是你所需要的一切 (https://arxiv.org/abs/1706.03762)》论文、Transformer博客文章（Transformer：一种用于语言理解的新型神经网络架构 (https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html)）以及Tensor2Tensor公告 (https://ai.googleblog.com/2017/06/accelerating-deep-learning-research.html)。

- Watch Łukasz Kaiser's talk (https://www.youtube.com/watch?v=rBCqOTEfxvg) walking through the model and its details

  观看卢卡斯·凯泽的演讲 (https://www.youtube.com/watch?v=rBCqOTEfxvg)，其中详细介绍了该模型及其细节

- Play with the Jupyter Notebook provided as part of the Tensor2Tensor repo (https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb)

  使用Tensor2Tensor代码库中提供的Jupyter笔记本进行实践

- Explore the Tensor2Tensor repo (https://github.com/tensorflow/tensor2tensor).
  探索Tensor2Tensor代码库 (https://github.com/tensorflow/tensor2tensor)。

Follow-up works: 后续工作：

- Depthwise Separable Convolutions for Neural Machine Translation

  用于神经机器翻译的深度可分离卷积

  (https://arxiv.org/abs/1706.03059)
- One Model To Learn Them All 一个模型学习所有内容 (https://arxiv.org/abs/1706.05137)
- Discrete Autoencoders for Sequence Models

  用于序列模型的离散自编码器

  (https://arxiv.org/abs/1801.09797)
- Generating Wikipedia by Summarizing Long Sequences

  通过总结长序列生成维基百科

  (https://arxiv.org/abs/1801.10198)
- Image Transformer 图像转换器 (https://arxiv.org/abs/1802.05751)
- Training Tips for the Transformer Model

  Transformer模型的训练技巧

  (https://arxiv.org/abs/1804.00247)
- Self-Attention with Relative Position Representations

  具有相对位置表示的自注意力机制

  (https://arxiv.org/abs/1803.02155)
- Fast Decoding in Sequence Models using Discrete Latent Variables

  使用离散潜变量的序列模型快速解码

  (https://arxiv.org/abs/1803.03382)
- Adafactor: Adaptive Learning Rates with Sublinear Memory Cost

  Adafactor：具有亚线性内存成本的自适应学习率

  (https://arxiv.org/abs/1804.04235)

Acknowledgements 致谢

Thanks to Illia Polosukhin (https://twitter.com/ilblackdragon), Jakob Uszkoreit (http://jakob.uszkoreit.net/), Llion Jones (https://www.linkedin.com/in/llion-jones-9ab3064b), Lukasz Kaiser (https://ai.google/research/people/LukaszKaiser), Niki Parmar (https://twitter.com/nikiparmar09), and Noam Shazeer (https://dblp.org/pers/hd/s/Shazeer:Noam) for providing feedback on earlier versions of this post.

感谢伊利亚·波洛苏欣（Illia Polosukhin） (https://twitter.com/ilblackdragon)、雅各布·乌兹科雷特（Jakob Uszkoreit） (http://jakob.uszkoreit.net/)、利昂·琼斯（Llion Jones） (https://www.linkedin.com/in/llion-jones-9ab3064b)、卢卡斯·凯泽（Lukasz Kaiser） (https://ai.google/research/people/LukaszKaiser)、尼基·帕尔马（Niki Parmar） (https://twitter.com/nikiparmar09)和诺姆·沙泽尔（Noam Shazeer） (https://dblp.org/pers/hd/s/Shazeer:Noam)对本文初稿提供的反馈。

Please hit me up on Twitter (https://twitter.com/JayAlammar) for any corrections or feedback.

如有任何更正或反馈，请在Twitter (https://twitter.com/JayAlammar)上联系我。
Written on June 27, 2018 写于2018年6月27日

Language Models & Co.

Large language models, their internals, and applications.

By Jay Alammar • Over 31,000 subscribers

Type your email... Subscribe

By subscribing you agree to Substack's Terms of Use, our Privacy Policy and our Information collection notice

substack

(http://creativecommons.org/licenses/by-nc-sa/4.0/)
This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (http://creativecommons.org/licenses/by-nc-sa/4.0/).

本作品采用知识共享署名-非商业性使用-相同方式共享4.0国际许可协议 (http://creativecommons.org/licenses/by-nc-sa/4.0/)进行许可。

Attribution example: 归因示例：
Alammar, J (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/ (https://jalammar.github.io/illustrated-transformer/)

Alammar, J（2018）。图解Transformer[博客文章]。取自https://jalammar.github.io/illustrated-transformer/ (https://jalammar.github.io/illustrated-transformer/)

Note: If you translate any of the posts, let me know so I can link your translation to the original post. My email is in the about page (/about).

注意：如果你翻译了任何帖子，请告诉我，这样我可以将你的翻译链接到原始帖子。我的电子邮件在关于页面 (/about)上。

(https://github.com/jalammar) (https://www.linkedin.com/in/jalammar) (https://www.twitter.com/jayalammar)